УДК 004.9
DOI https://doi.org/10.32782/tnv-tech.2023.5.8

РЕАЛІЗАЦІЯ ОБ'ЄКТНО-ОРІЄНТОВАНОЇ ПАРАДИГМИ ПРОГРАМУВАННЯ МОВОЮ С

Стовманенко В. О. *– аспірант*
Чорноморського національного університету імені Петра Могили
ORCID ID: 0009-0000-7211-6967

Мова програмування С дозволяє створювати застосунки, що працюють на майже кожній актуальній платформі від поширених x86-сумісних процесорів, до MIPS у PlayStation One, від ARM у більшості сучасних смартфонів, до POWER-10 на великих серверах. Компілятор цієї мови з'являється для платформи інколи раніше ніж сама платформа з'являється фізично. Простота мови де є відносно невелика кількість вбудованих конструкцій дозволяє просто реалізувати початкову версію компілятора для цільової платформи. Створення застосунків із використанням С легше ніж з використанням асемблера, де, на відміну від С немає поняття циклів, типів чи методів, уже значно спрощує розробку, проте все іще є достатньо складним процесом так С не має вбудованих механізмів для керування пам'яттю, роботи з помилками або повторним використанням коду, окрім як використання спільних функцій та методів.

З іншого боку існують більш високорівневі мови, які вимагають значної кількості засобів для своєї роботи (інтерпретатор, віртуальна машина, набір системних бібліотек, які оптимізовані для конкретної платформи, тощо), але суттєво спрощують розробку. Часто, інфраструктура певної мови прив'язана до певної апаратної чи програмної платформи і не може бути використана аналогічним чином у інших середовищах, до того ж усі додаткові засоби необхідні для виконання, забирають частину ресурсів пристрою, що обмежує клас пристроїв, де можна виконувати код написаний цією мовою.

Існує спосіб отримати зручність переваги коду на С, що виконується швидко та майже будь-якій апаратній платформі й писати значно більш простий код. Для цього можна використовувати дворівневу компіляцію, де код мовою С – буде проміжним представленням і буде компілюватися в машинні інструкції компілятором С. Подібні мови уже існують (Vala, V, Zig) і кожна з них представляє свій варіант трансляції та своє бачення ООП. У статті розглядається бібліотека, що реалізує функціонал потрібний для забезпечення ООП, без прив'язки до конкретної апаратної платформи та з можливістю легко інтегрувати створені компоненти у інші програмні рішення.

Ключові слова: *ООП, С, трансляція, виключення, обробка помилок, інтерфейс, поліморфізм, віртуальна таблиця методів, ABI, стабільність API.*

Stovmanenko V. O. Object oriented paradigm implementation in C programming language

C programming language allows to create application and libraries that could be used on almost any relevant hardware platform. It could be x86-compatible desktop processor, MIPS in PlayStation One, ARM chip in a smartphone or a POWER-10 on some big server. Compiler for this language is often created sooner then the platform itself becomes available as a piece of hardware. Language simplicity with a relatively small amount of key words and included functionality allows to implement a basic C compiler quite easy. This language bring a noticeable simplification to the code. It is often used when the interaction with hardware is required, but in a very specific way. Other contender for this role is assembly for concrete platform, but it lacks such concepts as function, type, struct of cycle. Assembly program is basically a plain list of instructions and some of the have labels that allow jumping to it. And code written in C could be compiled for different architecture, while code in assembly has to be rewritten for other platform. However, while C code is easier then assembler, it is still more difficult then, Python, for example. Code has to be organized in functions, there are no namespaces, memory management is manual, no error handling mechanism, etc.

Meanwhile there are various high level languages, that have all these features and allow programmer to concentrate on ideas and concepts of concrete field and not think about low-level stuff. All these feature come with a price. It could be either interpreter, virtual machine, like CLR for C#, or various system components that should be present in order for such application to work. Sometimes infrastructure is tied to concrete hardware of software platform and could

not be used in other places. Like microcontrollers, for example. It also makes it more difficult to integrate components from various technologies together. To call Python library from C#, developer has to create a C wrapper that will interact with the Python interpreter and marshall the data between Python Interpreter and CLR. It is also true with C++ and Java, for example. One have to write a wrapper that will interact with a JVM to use Java component from C++ instead of just calling a method or a function.

It is possible to have advantages of C code that works quite fast and almost everywhere (with recompilation, of course), but to write much simpler code that looks more familiar to a developer that works with a higher level language. One could use two-level compilation. First stage is to compile the source language to C representation and then, use C compiler to generate native binary. Such languages are already exist (Vala, V, Zig) and each of them have unique way of generating C code and own runtime library, necessary to provide primitives. Some of them could be easily ported to other architectures (like in Zig) and some not (GObject in Vala, which depends on Glib, not to confuse with Glibc). Sometimes it is easier to integrate them in other application and sometimes no. This articles describe OOP library for C language that allows code generation from higher level language and allows to have common way of working with virtual tables for polymorphic objects, type system, error handling, which have no dependencies other then C standard library and allows easy integration into other components that could call a C entry point.

Key words: *OOP, C, translation, exception, error handling, interface, polymorphism, virtual method table, ABI, API stability.*

Постановка проблеми. Об'єктно орієнтоване програмування (ООП) – це парадигма побудови програми, що визначає програму як набір об'єктів, які містять дані та логіку, що взаємодіють між собою [1]. Вона не прив'язана до конкретної мови програмування чи середовища і передбачає реалізацію сутностей, які реалізують такі поняття:

– Спадкування (Inheritance). Здатність сутності повторно використовувати поведінку (клас) та/або структуру (тип).

– Інкапсуляція (Encapsulation). Здатність сутності приховувати особливості структури та поведінки. Приховування передбачає можливість працювати з сутністю таким чином аби не було потреби знати про те, як саме виконуватиметься той чи інший метод та без розуміння того, яку структуру має ця сутність.

– Поліморфізм (Polymorphism). Здатність сутності змінювати поведінку успадкованого класу, але бути доступною для використання у якості об'єкту базового класу.

Для ясності варто визначити поняття «клас», що визначає набір та особливості поведінки сутності. Це поняття безпосередньо залежить від розуміння поняття «поведінка». В даному випадку під поведінкою мається на увазі сукупність точок входу, які можуть бути використані кодом, що працює з сутністю для виконання деякої дії. Різні мови по-різному це реалізують (виклик методу C++, відправлення повідомлення у Smalltalk, підстановка функції у Haskell). Мова може не бути об'єктно-орієнтованою для того щоб мати клас. Іншим важливим поняттям є «тип». Тип визначає з яких елементів складається сутність і те, де саме в пам'яті конкретної сутності знаходиться той чи інший елемент даних. Комбінація типу та класу дозволяє говорити про поняття об'єкту. Повноцінна реалізація трьох принципів наведених вище дозволяє говорити про реалізацію парадигми ООП.

Кожне з цих понять може бути реалізовано по-різному. Наприклад спадкування може бути розділеним (коли окремо можна успадкувати структуру, але не поведінку та навпаки) або спільним (не можна явно успадкувати щось одне без іншого). Інкапсуляція в статично типізованих мовах, що компілюються (наприклад C++) передбачає обмеження видимості частини класу чи типу для того щоб інші частини застосунку не могли робити припущення про те, як побудовано

об'єкт. В той же час для динамічних мов, таких як Python, це є значно меншою складністю так як немає потреби наперед визначати потрібний тип чи клас об'єкта для того аби ним скористатися. Поліморфізм може бути реалізований через таблицю віртуальних методів, як це робить C# або через композицію об'єктів, як це робить Go.

В той же час мова програмування *C* – це мова, яка використовує структурну парадигму до програмування [2]. Відсутність засобів автоматичного управління пам'яттю, безпеки типів та наявність арифметики вказівників роблять цю мову достатньо складною та такою, що вимагає високого рівня професійності розробника для того щоб написати швидку та надійну програму цієї мовою. В той же час, це проста мова, що передбачає реалізацію відносно не великого обсягу конструкцій управління. Тому це мова, яку відносно просто перенести на інші архітектури та операційні системи [3]. Вбудовані системи можуть отримати переваги *C* (мінімальне виділення пам'яті необхідне для роботи застосунку без використання сторонніх бібліотек та висока швидкодія з можливістю додавати вставки безпосередньо на асемблері для реалізація платформозалежних шматків коду) і в той же час застосунки на *C* можуть використовуватися у великих розрахункових центрах, де потрібна максимальна швидкодія та робота із апаратною складовою [4]. Тому можливість використовувати звичні та популярні підходи ООП може бути корисною під час реалізації того чи іншого засобу на будь-якій платформі.

Мета. Продемонструвати можливості реалізації підходів ООП мовою програмування *C*, що не є об'єктно-орієнтованою відповідно до потреб розробника.

Аналіз останніх досліджень і публікацій. Значна частина публікацій, що стосується принципів об'єктно-орієнтованого програмування була випущена в 1990-х роках і визначала теоретичну базу [5] для реалізації подібного підходу до програмування. Сучасні дослідження на кшталт джерел [6, 7] більше концентруються на використанні мов програмування у конкретних контекстах і реалізації певних підходів ООП на вже розвинених мовах програмування, на кшталт C++, Python чи JavaScript. Підходи до реалізації ООП стосуються або реалізації певних мов програмування та теорії компіляторів або теорії множин і більше стосуються розвитку математичного апарату. Дана стаття розглядає реалізацію визначених підходів ООП на наявній мові програмування, що не містить їх.

Виділення не вирішених раніше частин загальної проблеми. Існує декілька підходів до реалізації ООП на *C* [8, 9]. Різність у реалізації дозволяє приводити різні приклади того, як принципи визначені у вступі перенесені в код мовою *C*. Кожен з них містить позитивні та негативні моменти.

– Використання бібліотеки/заголовків, які реалізують об'єктно-орієнтований підхід. Прикладами таких є GObject або KObject. Вони надають реалізацію базових класів для об'єктів та надають допоміжні структури для створення класів та типів визначають підхід до того як працюватимуть об'єкти. Обидві визначають макроси для створення визначень структур, які мають спрощувати для розробника визначення структур (C structs), які потрібні для визначення класів. Прикладом таких макросів для GObject є [8].

```
G_DEFINE_TYPE (NamespaceTypeName, namespace_type_name, NAMESPACE, TYPE_NAME, BaseType);
```

NamespaceTypeName – написання типу простору імен та назви типу у стилі PascalCase, що визначає назву типу і класу (в GObject тип і клас поєднано).

namespace_type_name – написання типу простору імен та назви типу у стилі snake_case, що визначає префікс функцій, які будуть використовуватися.

NAMESPACE – окрема назва простору імен.

TYPE_NAME – назва типу.

BaseType – назва базового класу та типу, який буде використовувати.

Помилки у цих назвах призводять до не очевидних помилок компіляції або виконання. Проте, у разі правильного написання це дає змогу не писати вручну велику кількість визначень, яких очікує бібліотека. Всі вони визначені відповідно до певних вимог і явно не перевіряються компілятором. При тому основний акцент на тому, що програмісти будуть писати код вручну, тому визначається значна кількість допоміжних елементів, які спрощують визначення елементів потрібних для бібліотеки.

– Модифікації компілятора та/або препроцесора для розширення функціоналу. В залежності від рівня змін можна говорити про створення окремих мов програмування. Прикладами є C2, C3, SCOOP та навіть, до певної міри такі проєкти як *Objective-C* та *C++*. Усі ці приклади здебільшого зберігають зворотну сумісність з C але додають конструкції на рівні мови, що перевіряються компілятором (на відміну від не явних угод з попереднього пункту) для реалізації принципів ООП (*Objective-C* та *C++* реалізують дуже різні підходи до ООП залишаючись зворотно сумісними з C). Наприклад методи *C++* реалізовані як функції *C*, що приймають вказівник на this, як 1-ий параметр функції. Таким чином

```
class MyClass
{
public:
   intMyMethod (int param1, int param2);
}
```

Перетворюється на точку входу в *C* в стилі [10].

```
int CXX__NS__MyMethod(MyClass* this, int param1, int param2);
```

Або, в залежності від компілятора

```
int CXX__NS__MyMethod(int param1, int param2);
```

А вказівник на поточний об'єкт передається іншим способом (наприклад через стек). Для написання коду на подібній мові не важливо яким самим способом це буде реалізовано до тих пір поки зберігається єдиний підхід. Проте для роботи з подібним кодом з інших мов або зі звичайного *C* – це є критичним. І здебільшого це ускладнює роботу з виклику методу чи функції. Також, в залежності від складності перетворень, це може суттєво сповільнювати компіляцію (шаблони, templates, достатньо складна операція, що забирає більшу частину часу компіляції коду на C++).

– Реалізація окремого транслятора, який генерує код на *C*, що компілюється звичайним компілятором мови *C*. Прикладами подібного є мова Vala [11], що містить окремий транслятор, що генерує код на C, який використовує бібліотеку GObeject та об'єкту модель, яку надає ця бібліотека. Іншим прикладом є мова V. На відміну від Vala, V не використовує наявну бібліотеку для реалізації ООП. Підхід

більше нагадує той, що використовується в мові Go. V не використовує наявної бібліотеки для реалізації ООП, але використовує бібліотеку для «збирача сміття», що вбудовується в кожний виконуваний файл. Подібний підхід дає максимальний ступінь свободи для реалізації мови, дозволяє реалізувати стабільний інтерфейс доступний для інших застосунків, які можуть викликати методи на *C* (на відміну від попереднього пункту) та отримувати доступ до інших бібліотек на *C* або іншій мові, яку можна викликати з коду поточної мови (як і в попередньому пункті). Проте в даному випадку зневадження та проблеми з підходами, які пропонує транслятор можуть переривати нормальне виконання програми в неочікуваний для розробника спосіб. Якщо розробник не занурюється в код, який автоматично створюється, то деякі особливості роботи застосунку будуть не очевидними. Також існують проблеми зі зневадженням. Так як кінцевий компілятор не знає про оригінальну мову, то і в двійковому файлі буде тільки інформація про символи з коду на *C*, який може суттєво відрізнятися від коду оригінальною мовою. В залежності від обраного підходу мова може вимагати реалізації базових типів на *C* так і використовувати виключно інструменти які надаються самою мовою або компілятором. У ситуації з Vala, робота об'єктної системи повністю визначається бібліотекою GObejct і отримати щось інше не можна. Тому вибір підходу є дуже важливим і суттєво вплине на можливості об'єктної системи, що буде реалізованим.

Тому уже існують засоби, які реалізують ООП у більшій чи меншій мірі на *C*. Проте рішення, яке одночасно давало б змогу писати код на *C* безпосередньо, давало змогу реалізовувати принципи ООП без або з мінімальною кількістю підходів, що покладаються на не явний контракт і було зручним рішенням для автоматичної генерації коду немає. Кожне з рішень має негативні моменти пов'язані з поставленою ціллю. Тому в даній статті розглядається бібліотека для реалізації принципів ООП на *C*, яку можна використовувати для вирішення трьох поставлених задач.

Виклад основного матеріалу. Для того щоб мати змогу писати код безпосередньо на *C*, було реалізовано бібліотеку, яку можна використовувати у потрібних місцях програми і, водночас, використовувати як ціль для генерації коду з іншої мови, як це робить Vala, V, Nim або інші мови. Далі буде розглянуто реалізації кожного з принципів у бібліотеці [12] та наведено приклади того, як це реалізовано і як це має використовуватися. Бібліотека визначає систему типів з базовим класом та типом Object. Це структура такого роду

```
struct object_t {
unsigned short marker;
unsigned int ref_count;
void* state;
Type* type;
};
```

– marker – це константне значення, що дозволить виявити чи є дані під вказівником об'єктом (необхідно для захисту від пошкоджень пам'яті адже можлива передача не правильних даних через вказівник типу void*). Кожен об'єкт містить цей заголовок, який дозволяє його ідентифікувати як правильний. Негативною рисою подібного є необхідність витрачати додаткові 2 байти на кожен створений об'єкт.

– ref_count – це кількість посилань на поточний об’єкт. Використовується для управління пам’яттю і слугує індикатором необхідності звільнити пам’ять, яку використовує об’єкт.

– state – це вказівник на дані конкретного об’єкту або його структуру. Тип Object не визначає ніякої структури самостійно. Натомість він визначає підхід для зберігання даних між типами.

– type – вказівник на об’єкт, який описує тип та клас поточного об’єкта.

В коді на *C* всі об’єкти фактично одного типу Object і можна тільки визначити інше ім’я для того ж типу для зручності.

```
typedef struct object_t MyType;
```

Таким чином присвоєння такого виду буде правильним

```
MyType1* obj = (MyType2*)get_object_2();
```

і не викличе ніяких попереджень компілятора. Для того щоб убезпечитися від подібного реалізовано функцію, що перевіряє тип перед здійсненням присвоєння. Так як кожен вказівник можна перевірити (чи дійсно це об’єкт) і кожен об’єкт містить тип, завжди можна перевірити чи можна здійснювати присвоєння або передачу в якості параметра. Для подібних перевірок клас Type визначає метод

```
bool object_type_is_assignable_from(Type* this, Type* that);
```,

що дає змогу перевірити чи можна присвоїти змінній типу this значення об’єкту типу that.

Іншою важливою структурою, що забезпечує роботу системи типів є тип та клас Type. В ньому містяться посилання на таблицю віртуальних методів, функції для виділення пам’яті та звільнення конкретної структури класу та інформація про наявні методи (що не є обов’язковим для забезпечення існування ООП підходу, але є зручним інструментом).

Далі про реалізацію для забезпечення кожного принципу ООП.

– Наслідування. Спадкування структури здійснюється через роботу з вказівником state. Кожен тип може визначати свій тип структури, де будуть зберігати дані визначені конкретним типом. Під час реєстрації типу в системі типів (це відбувається під час створення першого об’єкту цього типу) визначається розмір цієї структури або 0, якщо її немає. Далі, під час створення об’єкту даного типу, система типів виділяє такий обсяг пам’яті, який дозволить зберегти по одному екземпляру кожної структури для кожного елементу ієрархії. В загальному випадку для того, щоб отримати стан конкретного об’єкту свого типу треба здійснювати такий набір операцій.

```
static struct _my_type_state_t* namespace_my_type_get_state(MyType* this)
    char* stateHandle = (char*)object_object_get_state(this);          //1
    Type* thisType = object_get_my_type_type();                        //2
    Type* baseType = object_type_get_base_type(thisType);              //3
    int baseSize = object_type_get_size(baseType);                     //4
    return (struct _my_type_state_t*)(stateHandle + baseSize);         //5
}
```

Спочатку код отримує вказівник state (входження 1). Код за межами бібліотеки не може змінювати значення цього вказівника безпосередньо, проте бібліотека типів надає доступ до даних, які там знаходяться. Від автора конкретного користувацького залежить чи робити визначення структури, що міститься в сегменті пам'яті, куди вказує state чи ні. Наступне входження (2) отримує входження поточного типу. Тут використовується функція для отримання конкретного типу не залежного від змінної тому що функцію визначено у файлі з визначенням цього класу. Входження 3 дозволяє отримати базовий клас для поточного типу. У якості оптимізації можна використовувати наперед відомий батьківський тип. У входженні 4 отримується розмір стану батьківського класу. Фактично це зсув у байтах, який необхідно здійснити аби отримати інформацію про структуру конкретного типу. П'яте входження здійснює цей зсув на потрібну кількість байт для того щоб отримати адресу першого байту структури конкретного типу. При тому для реалізації такого методу потрібно знати тип структури, яка використовується в якості стану для конкретного об'єкту, що дозволяє забезпечувати інкапсуляцію та приховувати певні особливості реалізації.

Спадкування класу здійснюється за рахунок можливості використання будь-яких не поліморфних методів батьківського класу, які є доступними у певному місці в коді. Для поліморфних об'єктів визначається таблиця віртуальних функцій про яку більше в секції про поліморфізм.

– Інкапсуляція. *C* має достатньо можливостей для інкапсуляції даних навіть без урахування принципів ООП. Для публічного доступу достатньо використовувати заголовок з публічними типами та функціями, а у файлі з реалізаціями можна використовувати будь-які не зазначені у публічному інтерфейсі типи та функції. Можна визначити тільки назву типу не розкриваючи подробиць її фактичної структури (у таких ситуаціях не можна буде створити таку структуру або привести вказівник до цього типу). Проте код що з нею працює не знає які поля конкретна структура має. Структура object_t визначена саме таким чином. Її структура відома тільки в межах бібліотеки і не є публічно доступною. Таким чином визначення об'єкту є інкапсульованим. Можливо визначати функції для отримання та встановлення значень, які нададуть змогу працювати із конкретними значеннями без розкриття подробиць реалізації структури. За потреби автор типу все ж може розкривати інкапсуляцію. Для того аби дати доступ до частини або всієї структури типу достатньо визначити структуру в публічному заголовку для користування. Аналогічним чином можна визначати додаткові відомості, які мають бути доступними тільки для похідних типів. В бібліотеці визначено підхід, який дозволяє розмежовувати доступ до визначень класу підтримуючи різні рівні доступу. Кожен клас містить теку, в якій міститься клас із реалізаціями з розширенням «.c» та набір заголовків. Заголовок public.h містить усі визначення, які є публічними і є доступними для усіх хто використовує цей тип. За потреби можна визначати заголовок з назвою protected.h, який міститиме визначення, які мають бути доступні похідним типам. За потреби можна визначати будь-які інші заголовки з більш особливими обмеженнями.

– Поліморфізм. Для реалізації поліморфізму в структурі Type визначено поле void* vt. Робота з ним здійснюється аналогічним чином, як і з void* state у структурі object. Під час реєстрації вказується розмір таблиці віртуальних методів для конкретного класу і для її отримання потрібно змістити вказівник на таблицю на потрібну кількість байт та привести до потрібного типу. Після того можна викликати потрібний метод, вказівник на який збережено в таблиці. Варто зауважити, що таблиця в даному випадку це структура на кшталт наступної

```
struct _object_vt {
    String* (*to_string)(Object* this, Exception** exception);
    int (*hash)(Object* this, Exception** exception);
};
```

Враховуючи, що кожне поле подібної структури – це вказівник, то дану структуру можна представляти в вигляді масиву вказівників (у найбільш простому випадку void**). Індексом такого масиву може слугувати числовий ідентифікатор методу. На рівні коду на асемблері звертання до будь-якого поля структури – це читання адреси структури з певним зсувом. Тому в даному випадку можна говорити про таблицю. Під час реєстрації типу можна отримати вказівник на таблицю віртуальних методів та перевизначити входження і змінити входження. Батьківський клас при цьому не зазнає ніяких змін. Єдиною вимогою до нього є використання цієї ж таблиці віртуальних функцій для виклику. Таким чином виклик функції батьківського класу переспрямовується через таблицю у реалізацію потрібного класу. Після перевантаження функції все іще існує можливість викликати реалізацію базового класу через таблицю віртуальних функцій базового класу.

Додатково реалізовано обгортки для фундаментальних типів (цілі числа зі знаком та без знаку, дробові числа, символи та логічні значення), типи, які можна узагальнювати (підтримано дві стратегії для використання подібних класів, коли існує можливість використовувати скомпільовану версію, де підставлено потрібний тип і використання є оптимізованим і загальну версію, де використовується Object замість конкретних типів), інтерфейси та виключення (які не вимагають розгортання стеку, а є звичайною структурою, яка повертається як додатковий аргумент методу). Типова програма на *C*, що використовує можливості ООП виглядає так:

```
#include «object.h»
#include <stdio.h>
#include <stdlib.h>

int main() {
object_init();

Exception* exception = NULL;
IntArray* intArr = object_int_array_new(2, &exception);
if(exception != NULL) {
    String* msg = object_exception_get_message(exception);
    char* buffer = object_string_get_buffer(msg);
    printf(«%s\n», buffer);
    return 1;
}

object_int_array_set(intArr, 0, 2, &exception);
if(exception != NULL) {
    String* msg = object_exception_get_message(exception);
    char* buffer = object_string_get_buffer(msg);
    printf(«%s\n», buffer);
```

```
    return 1;
}

int val = object_int_array_get(intArr, 0, &exception);
if(exception != NULL) {
    String* msg = object_exception_get_message(exception);
    char* buffer = object_string_get_buffer(msg);
    printf(«%s\n», buffer);
    return 1;
}

printf(«Array value is %d\n», val);

Type* t = object_object_get_type(intArr);
while(t != NULL) {
    String* name = object_type_get_name(t);
    char* buffer = object_string_get_buffer(name);
    printf(«Type is %s\n», buffer);
    t = object_type_get_base_type(t);
}

BoxedInt32* boxed = object_boxed_int32_new(12);
int value = object_boxed_int32_get_value(boxed);
printf(«Boxed value is %d\n», value);

String* stringValue = object_object_to_string(boxed, &exception);
if(exception != NULL) {
    String* name = object_type_get_name(t);
    char* buffer = object_string_get_buffer(name);
    printf(«Exception is %s\n», buffer);
    t = object_type_get_base_type(t);
}

char* buf = object_string_get_buffer(stringValue);
printf(«Boxed string value is %s\n», buf);
object_object_unref(stringValue);
}
```

Реєстрація похідного типу виглядає так

```
#include «Core.Exception/protected.h»
#include «Core.String/public.h»

#include «string.h»
#include «stdlib.h»

static Type* objectArgumentExceptionType = NULL;

Type* object_get_argument_exception_type() {
```

```
    if(objectArgumentExceptionType != NULL) {
        return objectArgumentExceptionType;
    }

    String* name = object_string_new_with_data_owned(«Core.ArgumentException»);
    Type* baseType = object_get_exception_type();

    TypeSubstitution* subs[1];
    subs[0] = NULL;

    Method* methods[1];

    methods[0] = NULL;

    void* baseVT = object_type_get_virtual_table(baseType);
    ObjectVT* vt = (ObjectVT*)malloc(sizeof(ObjectVT));
    memcpy(vt, baseVT, sizeof(ObjectVT));

    TypeOrGenericParam* typeParameters[1];
    typeParameters[0] = NULL;

    TypeRestriction* restrictions[1];
    restrictions[0] = NULL;

    Type* interfaces[1];
    interfaces[0] = NULL;

    objectArgumentExceptionType = object_type_new(name, baseType, typeParameters,
subs, methods, vt, 0, 0, restrictions, interfaces);

    return objectArgumentExceptionType;
    }
```

Реалізований підхід дає схожий рівень інкапсуляції та швидкодії базових операцій, що і GObject (цю бібліотеку обрану як альтернативу для створення застосунків у просторі користувача, яка використовується як для написання коду на *C* безпосередньо так і для перетворення з іншої мови на *C* з використанням системи типів). На відміну від GObject, Obj (тимчасова назва розробленої бібліотеки), вимагає єдиного входження, яке має бути реалізовано відповідно до неявного контракту (функція що повертає інформацію про тип). При тому вона не має відповідати конкретному шаблону імені. Решта функція може бути довільною, проте об'єкт типу Type має бути правильно заповненим. Також Obj надає більше гнучкості для інкапсуляції. GObject вимагає публічного визначення віртуальної таблиці методів. GObject містить обмежену підтримку узагальнень. На рівні коду *C* узагальнень не існує, тому загальним підходом до узагальнень там є використання окремого поля, що утримує параметр типу та структури GValue, що описує узагальнені значення. При тому інформація про тип ніяк не описує узагальнений тип. Це критично для рішень на кшталт бібліотек серіалізації/десеріалізації (створення представлення об'єкту у певній формі/відтворення об'єкту на основі певного представлення).

Через історичні особливості не всі об'єкти містять тип. Наприклад звичайний масив не містить типу і не є об'єктом. В той же час GObject – це перевірена бібліотека, що використовується у великій кількості реалізованих продуктів та компонентів. З її використанням побудовано бібліотеку для створення графічних інтерфейсів користувача GTK, бібліотеку для роботу з растровою графікою Cairo, бібліотеку для обробки мультимедіа GStreamer тощо. Новий засіб використовує дещо інший підхід до структурного спадкування, розширює систему типів, додає більше даних про клас та тип і концентруються на можливості більш зручного створення коду з іншої мови при цьому забезпечуючи можливість писати код на *C* (з підходом аналогічним до динамічних мов на кшталт Python, коли є необхідність додавати перевірки типу на етапі виконання, що можна прибирати у оптимізованій версії збірки). При тому на поточному етапі він однозначно не є настільки надійним як GObject. Деякі можливості поки не реалізовано в бібліотеці. Наприклад, на поточному етапі не підтримано статичних елементів класу (не є архітектурним обмеженням та може бути додано), узагальнених інтерфейсів, а також можливості отримувати чи встановлювати значення структури відповідно до інформації про тип (reflection). Проте жодне із цих обмежень не є архітектурним і може бути реалізованим. Бібліотека написана з версією мови C99 та не має ніяких залежностей окрім як стандартної бібліотеки. Також є можливість прибрати і цю залежність, але для цього треба реалізувати частину функцій для маніпуляцією пам'яттю, перевіркою умов та буферами символів (c strings).

Висновки. В ході досліджень створено бібліотеку для мови програмування *C*, що реалізує ключові поняття об'єктно-орієнтованого програмування і визначає підходи для забезпечення функціонування об'єктної моделі. Проаналізовано наукові публікації, що аналізують можливості реалізації об'єктних підходів у програмуванні та наявні реалізації об'єктних моделей, які написано мовою програмування *C*. Реалізовано додатковий функціонал, який дозволить спростити розробку застосунків з використанням подібного засобу та дозволить реалізацію інших мов програмування, що будуть використовувати систему типів реалізованої бібліотеки. Визначено переваги нового засобу над наявними та наведено його недоліки.

Реалізована бібліотека демонструє можливості об'єктно-орієнтованого підходу, який можна використовувати на будь-якому оточенні та застосовувати як ціль для компіляції для іншої мови, що надає значно більш зручний синтаксис, але створює виконувані файли, що можуть використовувати наявні бібліотеки на *C* без штрафів до швидкодії для передачі значень між оточеннями та не вимагають інтерпретатора чи іншого оточення, що виконують проміжне представлення застосунку.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ:

1. Deborah J. Armstrong. The quarks of object-oriented development. *Communications of the ACM.* 2006. Vol. 49. № 2. p. 123–128.

2. Brian W. Kernighan, Dennis M. Ritchie. C Programming Language. Second edition. London, 1988. 272 p.

3. Rabinowitz Henry, Schaap Chaim. Portable C. Hoboken, 1990. 269 p.

4. Manuel Costanzo and others. Performance vs Programming Effort between Rust and C on Multicore Architectures: Case Study in N-Body. *Arxiv. 2021.* Vol 2107. Article 11912.

5. Booch, G. Object Oriented Analysis and Design with Applications. Boston, 1994. 608 p.

6. Iqbaldeep Kaur and others. Research paper on object oriented software engineering. *International journal of computer science and technology.* 2016. Vol. 7, Issue 4. p. 36–38.

7. H. Hourani, H. Wasmi, T. Alrawashdeh. A Code Complexity Model of Object Oriented Programming (OOP). *IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT).* Amman, Jordan, 2019, p. 560–564.

8. Репозиторій коду бібліотеки GLib. URL: https://github.com/GNOME/glib/tree/main/glib. (дата звернення: 13.12.2023).

9. Репозиторій коду ядра Linux. URL: https://github.com/torvalds/linux. (дата звернення 13.12.2023).

10. Репозиторії із стандартами C++. URL: https://github.com/cplusplus/draft (дата звернення 13.12.2023).

11. Репозиторій транслятора мови Vala. URL: https://gitlab.gnome.org/GNOME/vala (дата звернення 13.12.2023).

12. Репозиторій реалізованої бібліотеки. URL: https://github.com/FlaviusHouk/Obj (дата звернення 13.12.2023).

REFERENCES:

1. Armstrong, D. J. (2006). The quarks of object-oriented development. *Communications of the ACM*, *49*(2), 123–128.

2. Kernighan, B. W., & Ritchie, D. (1988, March 22). *C Programming Language*. Prentice Hall.

3. Rabinowitz, H., & Schaap, C. (1990, January 1). *Portable C*.

4. Costanzo, Rucci, Naiouf, & De Giusti. (2021, October). Performance vs Programming Effort between Rust and C on Multicore Architectures: Case Study in N-Body. *Arxiv*. Retrieved December 13, 2023, from https://arxiv.org/abs/2107.11912

5. Booch, G. (1994, January 1). *Object-oriented Analysis and Design with Applications*. Addison-Wesley Professional.

6. Kaur, Kaur, Ummat, Kaur, & Kaur. (2016, October). Research Paper on Object Oriented Software Engineering. *International Journal of Computer Science and Technology*, *7*(4), 36–38. http://www.ijcst.com/vol74/1/8-iqbaldeep-kaur.pdf

7. Hourani, H., Wasmi, H., & Alrawashdeh, T. (2019, April). A Code Complexity Model of Object Oriented Programming (OOP). *2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)*. https://doi.org/10.1109/jeeit.2019.8717448

8. GNOME Foundation. (n.d.). *GLib source code*. GitHub. Retrieved December 13, 2023, from https://github.com/GNOME/glib/tree/main/glib.

9. Linux Foundation. (n.d.). *Linux kernel source tree*. GitHub. Retrieved December 13, 2023, from https://github.com/torvalds/linux.

10. *C++ standards drafts*. (n.d.). GitHub. Retrieved December 13, 2023, from https://github.com/cplusplus/draft

11. GNOME Foundation. (n.d.). *Vala language tranlator*. GitLab. Retrieved December 13, 2023, from https://gitlab.gnome.org/GNOME/vala.

12. Vladyslav Stovmanenko. *C Obj library*. GitHub, Retrieved December 13, 2023, from: https://github.com/FlaviusHouk/Obj.